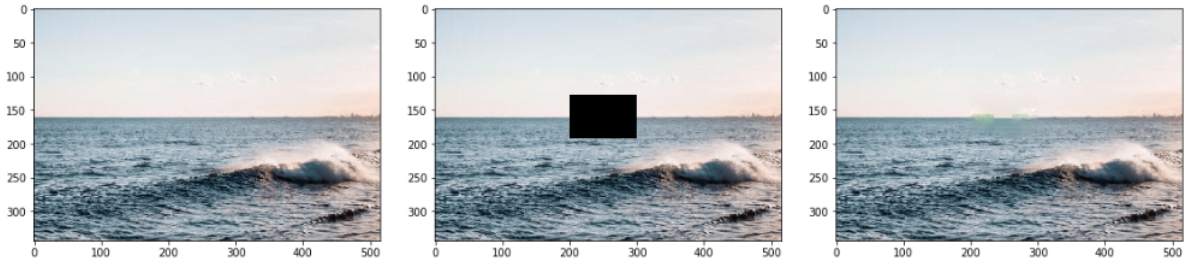
Saad El Madafri
Louis Gavalda
Coline Lacoux

Projet
Inpainting



Machine Learning

M1 ANDROIDE
Sorbonne Université

SCIENCES SORBONNE UNIVERSITÉ

2020

Indroduction

L'objectif de notre projet est d'utiliser des outils de Machine Learning dans le traitement d'images, en particulier l'*inpainting*. L'*inpainting* s'attache à la reconstruction d'image bruitées, ou dont il manque une partie. Pour cela, nous allons implémenter une technique présentée dans l'article [1] utilisant l'algorithme du LASSO.

Nous avons tout d'abord comparé trois algorithmes de régression linéaire pour saisir l'intérêt de l'algorithme du LASSO. Ensuite, nous avons implémenté des outils de traitement d'image et de reconstitution de pixels manquants basés sur cet algorithme. Pour finir, nous vous proposons dans la partie 2.3 différentes heuristiques pour remplir une image de manière intelligente.

Table des matières

Partie 1

Préambule : régression linéaire, régression ridge et LASSO

Etant donné un ensemble d'apprentissage $E = \{(x^i, y^i) \in \mathbb{R} \times \mathbb{R}\}$, la régression linéaire s'attache à trouver une fonction linéaire $f_w(x) = w_0 + \sum_{i=1}^{d} w_i x_i$ de vecteur de poids $w \in \mathbb{R}^{d+1}$ qui minimise l'erreur aux moindres carrés :

$$\text{MSE}(f_w, E) = \frac{1}{n} \sum_{i=0}^{n} (y^i - \omega_0 - \sum_{j=1}^{d} \omega_j x_j^i)^2$$

Cette méthode peut être adaptée à la classification binaire, en considérant les labels dans $Y = \{-1, +1\}$.

Deux autres variantes, issues de cette méthode, qui considèrent une version pénalisée du coût :

— la régression ridge (ou régularisation de Tikhonov, ou régularisation L_2) qui considère une pénalisation par la norme 2 de w :

$$L_2(f_w, E) = \text{MSE}(f_w, E) + \alpha \|w\|_2^2$$

avec $\|w\|_2^2 = \sum_{i=0}^{d} |w_i|$

— l'algorithme du LASSO qui considère une pénalisation par la norme 1 de w :

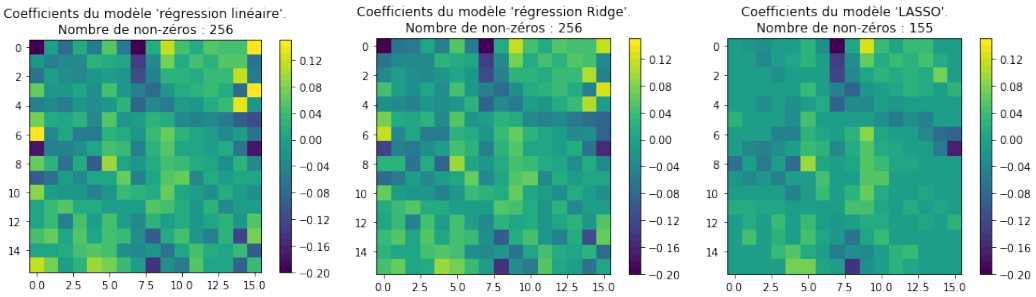
$$L_1(f_w, E) = \text{MSE}(f_w, E) + \alpha \|w\|_1$$

avec $\|w\|_1 = \sum_{i=0}^{d} |w_i|$

Nous avons comparé ces trois modèles sur le jeu de données USPS (classification de chiffres manuscrits de 1 à 9) en effectuant une validation croisée sur l'ensemble d'apprentissage. Les trois modèles ont donné des résultats similaires en précision de score : ≈ 0.97 en apprentissage et ≈ 0.94 en test.

Cependant on observe des différences sur les vecteurs de poids obtenu. Sur la figure 1.1, on peut voir que la régression ridge régule les coefficients. En effet, on

voit que, par rapport à la régression classique, certains coefficients se sont rapprochés de 0, ce qui réduit leur impact dans la classification. Quand au vecteur de poids obtenus par l'algorithme du LASSO, on remarque que certains poids sont réduits à zéro, donc certains pixels ne sont pas du tout pris en compte dans la prédiction : cela est dû à la régularisation L_2 qui pénalise les poids faibles.



(a) Régression linéaire (b) Régression Ridge (c) Algorithme du Lasso

FIGURE 1.1 – Vecteurs poids des différents modèles linéaires

L'algorithme du LASSO effectue donc une "sélection de features". Il ne conserve que les variables les plus pertinentes lors de son interprétation. Ici il y en a 155.

Partie 2

LASSO et Inpainting

2.1 Formalisation

Cette partie est consacrée à l'implémentation de la technique d'inpainting présentée dans [1] utilisant le Lasso et sa capacité à trouver une solution parcimonieuse en termes de poids. Ces travaux sont inspirés des recherches en Apprentissage de dictionnaire et représentation parcimonieuse pour les signaux [3].

Nous appellerons un *patch* un petit carré de l'image de longueur h. L'objectif est de réussir à construire un dictionnaire d'atomes - les patchs élémentaires - et trouver un algorithme de décomposition/recomposition de patchs à partir des atomes du dictionnaire, i.e. trouver des poids parcimonieux sur chaque patch tels que la combinaison linéaire des atomes permette d'approcher le patch d'origine.

2.2 Outils développés

Notre projet a été développé en Python 3.

Nous avons codé les fonctions pour le traitement d'images et la manipulation de patchs et de dictionnaires dans le fichier `code_python/tools.py`.

Le fichier `algorithms.py` contient les implémentations des différentes heuristiques qui seront présentées dans la partie 2.3.

Les tests de nos fonctions et nos expériences ont été réalisés dans le notebook `Projet_Inpainting.ipynb`.

2.3 Algorithmes de reconstruction

2.3.1 Algorithme "naïf"

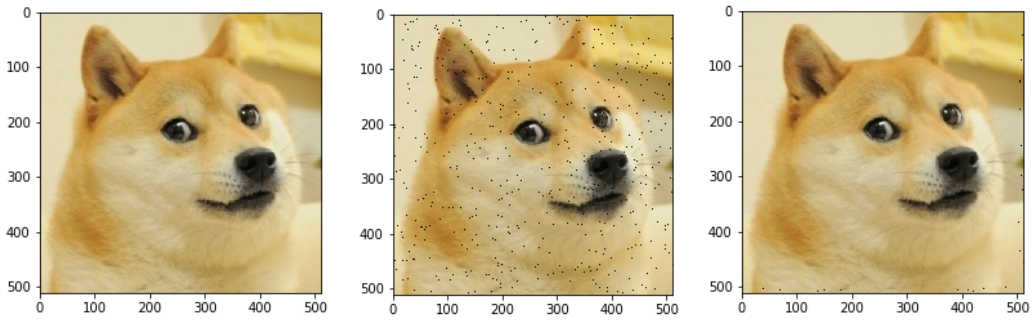
Notre premier algorithme `naive_reconstruction()` reconstruit naïvement une image en remplaçant les patchs, dont il manque des pixels, sans ordre déterminé. Si un patch ne contient aucun pixel exprimé, alors il n'est pas reconstruit.

Cette méthode est efficace sur les images bruitées comme on peut le voir sur la figure 2.2, à condition que la largeur des patchs lors du découpage soit assez petite afin qu'il puisse y avoir des patchs sans pixels manquants pour remplir le dictionnaire

(a) Patch bruité (b) Patch débruité

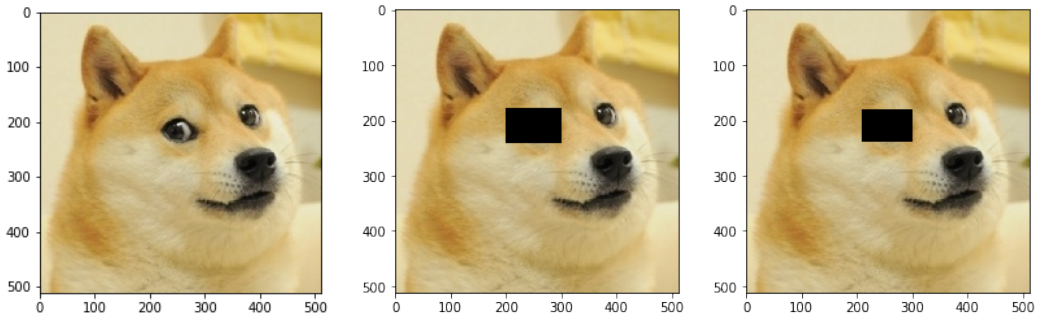
FIGURE 2.1 – Exemple de débruitage sur un patch

des atomes, nécessaire à l'apprentissage. Mais elle n'est pas efficace lorsque toute une zone de l'image est manquante, car les patchs au centre de la zone ne contiennent pas de pixels exprimés et ne pourront pas être reconstruits, comme on le voit sur la figure 2.3 où seuls les bords du rectangle sont restitués.



(a) Shiba d'origine (b) Shiba bruité (c) Shiba débruité

FIGURE 2.2 – Exemple de débruitage sur un Shiba Inu

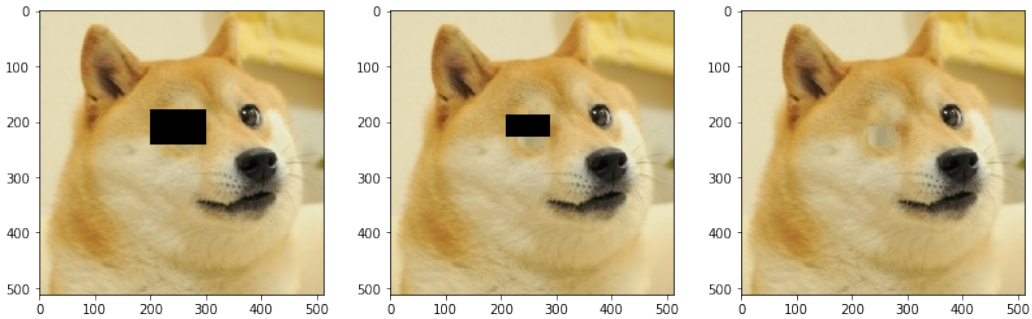


(a) Shiba d'origine (b) Shiba bruité (c) Shiba débruité

FIGURE 2.3 – Exemple de débruitage incomplet

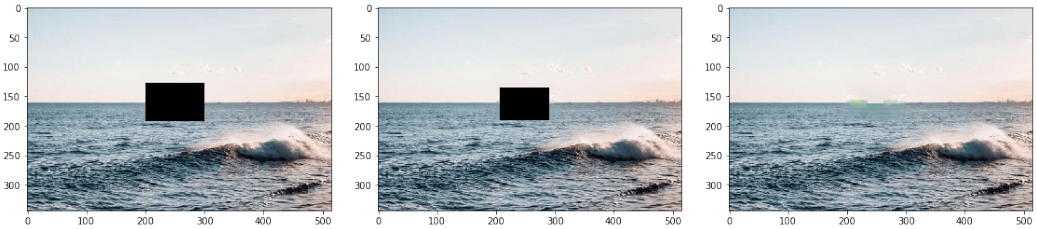
2.3.2 Algorithme "onion peel"

Notre second algorithme est basé sur la stratégie "onion peel" : On commence par remplir la partie manquante de l'image en centrant les patchs sur les bords de celle-ci. On remplit le contour de la partie manquante, puis on met à jour le dictionnaire des atomes et on recommence. Ainsi, on reconstruit l'image au fur et à mesure vers le centre de la partie manquante, en procédant par couches. C'est ainsi qu'on a pu reconstituer le pelage du Shiba Hinu à la place de son oeil, voir figure 2.4.



(a) Shiba endommagé (b) Etape intermédiaire (c) Résultat final

FIGURE 2.4 – Etapes de la reconstitution par "pelure d'oignon"



(a) Paysage endommagé (b) Etape intermédiaire (c) Résultat final

FIGURE 2.5 – Exemple sur un paysage de mer

Cette stratégie est efficace pour reconstituer une zone qui contient une seule texture, mais elle l'est moins lorsque la partie de l'image endommagée contient un changement de texture. En effet, on remarque sur la reconstitution du paysage marin que la frontière entre l'eau et le ciel est floue.

Cela est dû au fait qu'on met à jour à chaque nouvelle couche le dictionnaire des atomes avec les patchs reconstitués, ce qui introduit du bruit, et donc chaque nouvelle couche sera moins précise que la précédente. Cela ne change pas grand chose pour les zones de texture lisse, mais cette perte de précision tend à flouter les bordures.

2.3.3 Algorithme "structure preservation"

Pour palier au problème soulevé dans la partie 2.3.2, nous nous somme interessé à une stratégie énoncée dans l'article [2].

L'idée est de remplir en priorité les patchs ayant le plus de pixels exprimés, ou qui sont dans la continuité d'une structure, autrement dit qui contient des bordures, afin de préserver les structures en les traitant avant d'ajouter du bruit aux patchs d'apprentissage.

Pour cela, nous calculons le dictionnaire des patchs à remplir comme dans l'agorithm "onion peel", mais cette fois-ci nous assignons à chaque patch une valeur de priorité de remplissage que nous nommons *prior*, et nous traiterons les patchs selon leur ordre de priorité. Nous avons calculé la priorité de la manière suivante :

$$prior = \mathrm{C}(patch) * \mathrm{D}(patch)$$

avec :

— $\mathrm{C}(patch) = confidence$ du patch, i.e le nombre de pixels exprimés dans le patch divisé par 2.

— $\mathrm{D}(patch) = data\_term$ du patch, i.e le nombre de pixels du patch faisant partie d'une bordure. Les bordures sont identifiées par calcul du gradient sur la matrice de l'image (exemple figure 2.6).

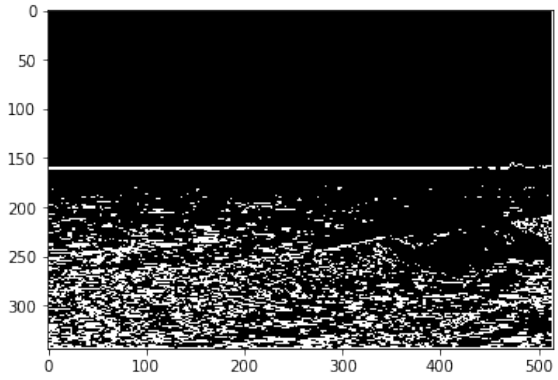
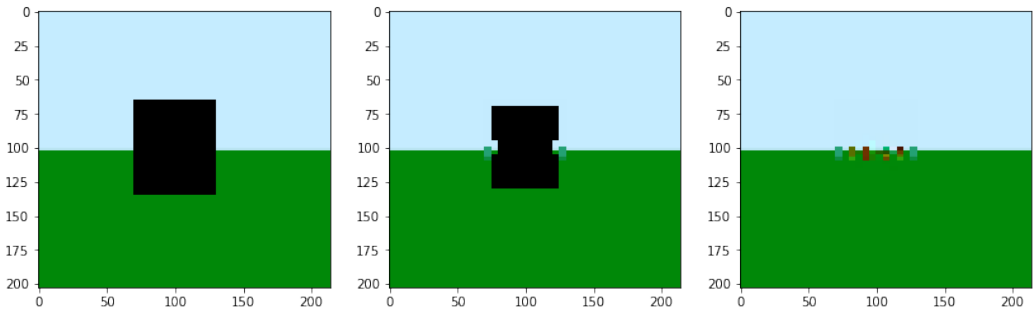


FIGURE 2.6 – Détection de contours de l'image paysage.png

Une fois que le dictionnaire des patchs à remplir est vide, on recentre les patchs et on recalcule les valeurs *prior* de chaque patch, puis on recommence. On continue jusqu'à n'avoir plus aucun patch à remplir.

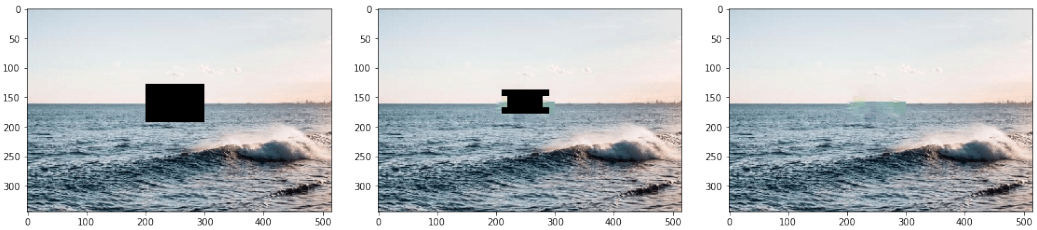
L'algorithme fonctionne bien sur des exemples simples, comme la figure 2.7.

Mais cela n'améliore pas autant que souhaité la reconstitution. En effet, l'algorithme possède ses limites car la reconnaissance de bords est facilement bruitée par des textures trop hétérogènes comme l'eau, comme on peut le voir sur la figure 2.8.



(a) Image endommagée (b) Etape intermédiaire (c) Résultat final

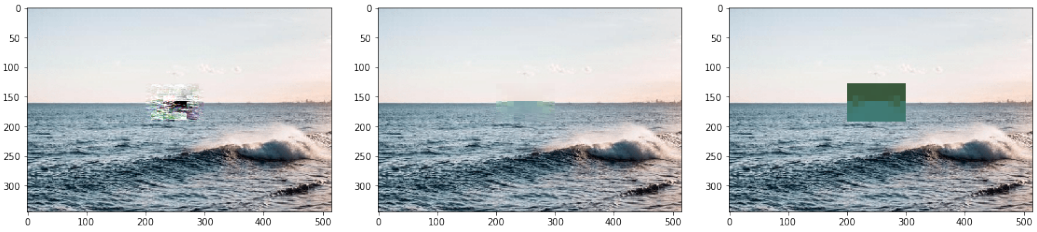
FIGURE 2.7 – Résultat sur un exemple simple



(a) Paysage endommagée (b) Etape intermédiaire (c) Résultat final

FIGURE 2.8 – Résultat sur le paysage de mer

2.4 L'impact du paramètre alpha



(a) alpha = 0 (b) alpha = 0.005 (c) alpha = 1

FIGURE 2.9 – Résultat pour différentes valeurs de alpha

La valeur de alpha est déterminante dans notre algorithme car elle contrôle le nombre de composantes non nulles dans le vecteur de coefficients du modèle.

Ainsi, si alpha est trop petit, trop de patchs sont pris en compte pour reconstituer les pixels manquants et on obtient des résultats incohérents. Par exemple, sur la figure 2.9a, on retrouve des morceaux de vague dans la mer calme car des patchs contenant des morceaux de vagues ont été considérés comme pertinents dans la reconstruction.

Si au contraire alpha est trop grand, aucun patch est pris en compte. Les pixels manquants sont remplacés par une valeur moyenne (moyenne de l'image applatie) comme on peut le voir sur la figure 2.9c.

Il est donc important de choisir une bonne valeur de alpha. Par exemple, pour

notre image `paysage.png`, $alpha = 0.005$ est un bon choix car il restitue plutôt bien l'image, comme on peut le voir sur la figure 2.9b.

2.5 Conclusion

Cette technique de reconstruction d'images sur la base de l'algorithme du LASSO est efficace pour reconstruire des parties d'images manquantes, tout en essayant de préserver les bordures comme dans la partie 2.8, ou pour supprimer une partie d'une image comme l'oeil du Shiba dans la partie 2.4. Un des inconvénients est le temps d'exécution de nos algorithmes : certaines images mettent plusieurs minutes à se reconstruire. Plus alpha diminue, plus l'algorithme est long à converger donc plus le nombre d'itérations nécessaires est grand. C'est pourquoi la cross-validation qui permet de choisir le meilleur alpha peut prendre du temps à s'exécuter. De plus, le temps d'exécution augmente fortement quand on diminue la valeur *step* (pas d'échantillonage) ou h (largeur des patchs), or cette dernière est déterminante pour la précision de la reconstruction obtenue.

Bibliographie

[1] Bin Shen et al. “Image inpainting via sparse representation”. In : *2009 IEEE International Conference on Acoustics, Speech and Signal Processing*. Taipei, Taiwan : IEEE, avr. 2009.

[2] A. Criminisi, P. Perez et K. Toyama. “Region Filling and Object Removal by Exemplar-Based Image Inpainting”. In : *IEEE Transactions on Image Processing* 13 (sept. 2004).

[3] Julien Mairal. *Sparse coding and Dictionnary Learning for Image Analysis*. 2010.